类和结构体（Classes and Structures）

> 1.0 翻译：JaySurplus (https://github.com/JaySurplus) 校对：sg552 (https://github.com/sg552)
>
> 2.0 翻译+校对：SkyJean (https://github.com/SkyJean)
>
> 2.1 校对：shanks (http://codebuild.me)，2015-10-29
>
> 2.2 校对：SketchK (https://github.com/SketchK) 2016-05-13

本页包含内容：

*类*和*结构体*是人们构建代码所用的一种通用且灵活的构造体。我们可以使用完全相同的语法规则来为类和结构体定义属性（常量、变量）和添加方法，从而扩展类和结构体的功能。

与其他编程语言所不同的是，Swift 并不要求你为自定义类和结构去创建独立的接口和实现文件。你所要做的是在一个单一文件中定义一个类或者结构体，系统将会自动生成面向其它代码的外部接口。

> 注意
> 通常一个 类 的实例被称为 对象 。然而在 Swift 中，类和结构体的关系要比在其他语言中更加的密切，本章中所讨论的大部分功能都可以用在类和结构体上。因此，我们会主要使用 实例 而不是 对象 。

类和结构体对比

Swift 中类和结构体有很多共同点。共同处在于：

- 定义属性用于存储值
- 定义方法用于提供功能
- 定义附属脚本用于访问值
- 定义构造器用于生成初始化值
- 通过扩展以增加默认实现的功能
- 实现协议以提供某种标准功能

更多信息请参见属性 (./10_Properties.html)，方法 (./11_Methods.html)，下标 (./12_Subscripts.html)，构造过程 (./14_Initialization.html)，扩展 (./21_Extensions.html)，和协议 (./22_Protocols.html)。

与结构体相比，类还有如下的附加功能：

- 继承允许一个类继承另一个类的特征
- 类型转换允许在运行时检查和解释一个类实例的类型
- 析构器允许一个类实例释放任何其所被分配的资源
- 引用计数允许对一个类的多次引用

更多信息请参见继承 (./13_Inheritance.html)，类型转换 (./19_Type_Casting.html)，析构过程 (./15_Deinitialization.html)，和自动引用计数 (./16_Automatic_Reference_Counting.html)。

> 注意
> 结构体总是通过被复制的方式在代码中传递，不使用引用计数。

定义语法

类和结构体有着类似的定义方式。我们通过关键字 `class` 和 `struct` 来分别表示类和结构体，并在一对大括号中定义它们的具体内容：

```
class SomeClass {
    // class definition goes here
}
struct SomeStructure {
    // structure definition goes here
}
```

关于 (http://wiki.jikexueyuan.com/project/swift/)

欢迎使用 Swift (http://wiki.jikexueyuan.com/project/swift/chapter1/chapter1.html)

Swift 教程 (http://wiki.jikexueyuan.com/project/swift/chapter2/chapter2.html)

基础部分 (http://wiki.jikexueyuan.com/project/swift/chapter2/01_The_Basics.html)

基本运算符 (http://wiki.jikexueyuan.com/project/swift/chapter2/02_Basic_Operators.html)

字符串和字符 (http://wiki.jikexueyuan.com/project/swift/chapter2/03_Strings_and_Characters.html)

集合类型 (http://wiki.jikexueyuan.com/project/swift/chapter2/04_Collection_Types.html)

Via 由 [极客学院 Wiki (http://wiki.jikexueyuan.com)] 提供

> 注意
>
> 在你每次定义一个新类或者结构体的时候，实际上你是定义了一个新的 Swift 类型。因此请使用 `UpperCamelCase` 这种方式来命名（如 `SomeClass` 和 `SomeStructure` 等），以便符合标准 Swift 类型的大写命名风格（如 `String`， `Int` 和 `Bool` ）。相反的，请使用 `lowerCamelCase` 这种方式为属性和方法命名（如 `framerate` 和 `incrementCount` ），以便和类型名区分。

以下是定义结构体和定义类的示例：

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

在上面的示例中我们定义了一个名为 `Resolution` 的结构体，用来描述一个显示器的像素分辨率。这个结构体包含了两个名为 `width` 和 `height` 的存储属性。存储属性是被捆绑和存储在类或结构体中的常量或变量。当这两个属性被初始化为整数 `0` 的时候，它们会被推断为 `Int` 类型。

在上面的示例中我们还定义了一个名为 `VideoMode` 的类，用来描述一个视频显示器的特定模式。这个类包含了四个变量存储属性。第一个是 *分辨率* ，它被初始化为一个新的 `Resolution` 结构体的实例，属性类型被推断为 `Resolution` 。新 `VideoMode` 实例同时还会初始化其它三个属性，它们分别是，初始值为 `false` 的 `interlaced` ，初始值为 `0.0` 的 `frameRate` ，以及值为可选 `String` 的 `name` 。 `name` 属性会被自动赋予一个默认值 `nil` ，意为"没有 `name` 值"，因为它是一个可选类型。

类和结构体实例

`Resolution` 结构体和 `VideoMode` 类的定义仅描述了什么是 `Resolution` 和 `VideoMode` 。它们并没有描述一个特定的分辨率（resolution）或者视频模式（video mode）。为了描述一个特定的分辨率或者视频模式，我们需要生成一个它们的实例。

生成结构体和类实例的语法非常相似：

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

结构体和类都使用构造器语法来生成新的实例。构造器语法的最简单形式是在结构体或者类的类型名称后跟随一对空括号，如 `Resolution()` 或 `VideoMode()` 。通过这种方式所创建的类或者结构体实例，其属性均会被初始化为默认值。构造过程 (./14_Initialization.html)章节会对类和结构体的初始化进行更详细的讨论。

属性访问

通过使用*点语法*（*dot syntax*），你可以访问实例的属性。其语法规则是，实例名后面紧跟属性名，两者通过点号(`.`)连接：

```
print("The width of someResolution is \(someResolution.width)")
// 输出 "The width of someResolution is 0"
```

在上面的例子中， `someResolution.width` 引用 `someResolution` 的 `width` 属性，返回 `width` 的初始值 `0` 。

你也可以访问子属性，如 `VideoMode` 中 `Resolution` 属性的 `width` 属性：

```
print("The width of someVideoMode is \(someVideoMode.resolution.width)")
// 输出 "The width of someVideoMode is 0"
```

你也可以使用点语法为变量属性赋值：

```
someVideoMode.resolution.width = 1280
print("The width of someVideoMode is now \(someVideoMode.resolution.width)")
// 输出 "The width of someVideoMode is now 1280"
```

> 注意
>
> 与 Objective-C 语言不同的是，Swift 允许直接设置结构体属性的子属性。上面的最后一个例子，就是直接设置了 `someVideoMode` 中 `resolution` 属性的 `width` 这个子属性，以上操作并不需要重新为整个 `resolution` 属性设置新值。

结构体类型的成员逐一构造器（Memberwise Initializers for Structure Types）

所有结构体都有一个自动生成的*成员逐一构造器*，用于初始化新结构体实例中成员的属性。新实例中各个属性的初始值可以通过属性的名称传递到成员逐一构造器之中：

```
let vga = Resolution(width:640, height: 480)
```

与结构体不同，类实例没有默认的成员逐一构造器。构造过程 (./14_Initialization.html)章节会对构造器进行更详细的讨论。

结构体和枚举是值类型

*值类型*被赋予给一个变量、常量或者被传递给一个函数的时候，其值会被*拷贝*。

在之前的章节中，我们已经大量使用了值类型。实际上，在 Swift 中，所有的基本类型：整数（Integer）、浮点数（floating-point）、布尔值（Boolean）、字符串（string)、数组（array）和字典（dictionary），都是值类型，并且在底层都是以结构体的形式所实现。

在 Swift 中，所有的结构体和枚举类型都是值类型。这意味着它们的实例，以及实例中所包含的任何值类型属性，在代码中传递的时候都会被复制。

请看下面这个示例，其使用了前一个示例中的 `Resolution` 结构体：

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

在以上示例中，声明了一个名为 `hd` 的常量，其值为一个初始化为全高清视频分辨率（ `1920` 像素宽， `1080` 像素高）的 `Resolution` 实例。

然后示例中又声明了一个名为 `cinema` 的变量，并将 `hd` 赋值给它。因为 `Resolution` 是一个结构体，所以 `cinema` 的值其实是 `hd` 的一个拷贝副本，而不是 `hd` 本身。尽管 `hd` 和 `cinema` 有着相同的宽（width）和高（height），但是在幕后它们是两个完全不同的实例。

下面，为了符合数码影院放映的需求（ `2048` 像素宽， `1080` 像素高）， `cinema` 的 `width` 属性需要作如下修改：

```
cinema.width = 2048
```

这里，将会显示 `cinema` 的 `width` 属性确已改为了 `2048`：

```
print("cinema is now  \(cinema.width) pixels wide")
// 输出 "cinema is now 2048 pixels wide"
```

然而，初始的 `hd` 实例中 `width` 属性还是 `1920`：

```
print("hd is still \(hd.width) pixels wide")
// 输出 "hd is still 1920 pixels wide"
```

在将 `hd` 赋予给 `cinema` 的时候，实际上是将 `hd` 中所存储的值进行拷贝，然后将拷贝的数据存储到新的 `cinema` 实例中。结果就是两个完全独立的实例碰巧包含有相同的数值。由于两者相互独立，因此将 `cinema` 的 `width` 修改为 `2048` 并不会影响 `hd` 中的 `width` 的值。

枚举也遵循相同的行为准则：

```
enum CompassPoint {
    case North, South, East, West
}
var currentDirection = CompassPoint.West
let rememberedDirection = currentDirection
currentDirection = .East
if rememberedDirection == .West {
    print("The remembered direction is still .West")
}
// 输出 "The remembered direction is still .West"
```

上例中 `rememberedDirection` 被赋予了 `currentDirection` 的值，实际上它被赋予的是值的一个拷贝。赋值过程结束后再修改 `currentDirection` 的值并不影响 `rememberedDirection` 所储存的原始值的拷贝。

类是引用类型

与值类型不同，引用类型在被赋予到一个变量、常量或者被传递到一个函数时，其值不会被拷贝。因此，引用的是已存在的实例本身而不是其拷贝。

请看下面这个示例，其使用了之前定义的 `VideoMode` 类：

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

以上示例中，声明了一个名为 `tenEighty` 的常量，其引用了一个 `VideoMode` 类的新实例。在之前的示例中，这个视频模式（video mode）被赋予了HD分辨率（ `1920 * 1080` ）的一个拷贝（即 `hd` 实例）。同时设置为 `interlaced` ，命名为 `“1080i”` 。最后，其帧率是 `25.0` 帧每秒。

然后， `tenEighty` 被赋予名为 `alsoTenEighty` 的新常量，同时对 `alsoTenEighty` 的帧率进行修改：

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

因为类是引用类型，所以 `tenEight` 和 `alsoTenEight` 实际上引用的是相同的 `VideoMode` 实例。换句话说，它们是同一个实例的两种叫法。

下面，通过查看 `tenEighty` 的 `frameRate` 属性，我们会发现它正确的显示了所引用的 `VideoMode` 实例的新帧率，其值为 `30.0` ：

```
print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// 输出 "The frameRate property of theEighty is now 30.0"
```

需要注意的是 `tenEighty` 和 `alsoTenEighty` 被声明为常量而不是变量。然而你依然可以改变 `tenEighty.frameRate` 和 `alsoTenEighty.frameRate` ，因为 `tenEighty` 和 `alsoTenEighty` 这两个常量的值并未改变。它们并不“存储”这个 `VideoMode` 实例，而仅仅是对 `VideoMode` 实例的引用。所以，改变的是被引用的 `VideoMode` 的 `frameRate` 属性，而不是引用 `VideoMode` 的常量的值。

恒等运算符

因为类是引用类型，有可能有多个常量和变量在幕后同时引用同一个类实例。（对于结构体和枚举来说，这并不成立。因为它们作为值类型，在被赋予到常量、变量或者传递到函数时，其值总是会被拷贝。）

如果能够判定两个常量或者变量是否引用同一个类实例将会很有帮助。为了达到这个目的，Swift 内建了两个恒等运算符：

- 等价于（ `===` ）
- 不等价于（ `!==` ）

运用这两个运算符检测两个常量或者变量是否引用同一个实例：

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same Resolution instance.")
}
//输出 "tenEighty and alsoTenEighty refer to the same Resolution instance."
```

请注意，“等价于”（用三个等号表示， `===` ）与“等于”（用两个等号表示， `==` ）的不同：

- “等价于”表示两个类类型（class type）的常量或者变量引用同一个类实例。
- “等于”表示两个实例的值“相等”或“相同”，判定时要遵照设计者定义的评判标准，因此相对于“相等”来说，这是一种更加合适的叫法。

当你在定义你的自定义类和结构体的时候，你有义务来决定判定两个实例“相等”的标准。在章节等价操作符(./25_Advanced_Operators.html#equivalence_operators)中将会详细介绍实现自定义“等于”和“不等于”运算符的流程。

指针

如果你有 C，C++ 或者 Objective-C 语言的经验，那么你也许会知道这些语言使用*指针*来引用内存中的地址。一个引用某个引用类型实例的 Swift 常量或者变量，与 C 语言中的指针类似，但是并不直接指向某个内存地址，也不要求你使用星号（ `*` ）来表明你在创建一个引用。Swift 中的这些引用与其它的常量或变量的定义方式相同。

类和结构体的选择

在你的代码中，你可以使用类和结构体来定义你的自定义数据类型。

然而，结构体实例总是通过值传递，类实例总是通过引用传递。这意味两者适用不同的任务。当你在考虑一个工程项目的数据结构和功能的时候，你需要决定每个数据结构是定义成类还是结构体。

按照通用的准则，当符合一条或多条以下条件时，请考虑构建结构体：

- 该数据结构的主要目的是用来封装少量相关简单数据值。
- 有理由预计该数据结构的实例在被赋值或传递时，封装的数据将会被拷贝而不是被引用。
- 该数据结构中储存的值类型属性，也应该被拷贝，而不是被引用。
- 该数据结构不需要去继承另一个既有类型的属性或者行为。

举例来说，以下情境中适合使用结构体：

- 几何形状的大小，封装一个 `width` 属性和 `height` 属性，两者均为 `Double` 类型。
- 一定范围内的路径，封装一个 `start` 属性和 `length` 属性，两者均为 `Int` 类型。
- 三维坐标系内一点，封装 `x`， `y` 和 `z` 属性，三者均为 `Double` 类型。

在所有其它案例中，定义一个类，生成一个它的实例，并通过引用来管理和传递。实际中，这意味着绝大部分的自定义数据构造都应该是类，而非结构体。

字符串(String)、数组(Array)、和字典(Dictionary)类型的赋值与复制行为

Swift 中，许多基本类型，诸如 `String`， `Array` 和 `Dictionary` 类型均以结构体的形式实现。这意味着被赋值给新的常量或变量，或者被传入函数或方法中时，它们的值会被拷贝。

Objective-C 中 `NSString`， `NSArray` 和 `NSDictionary` 类型均以类的形式实现，而并非结构体。它们在被赋值或者被传入函数或方法时，不会发生值拷贝，而是传递现有实例的引用。

> 注意
> 以上是对字符串、数组、字典的“拷贝”行为的描述。在你的代码中，拷贝行为看起来似乎总会发生。然而，Swift 在幕后只在绝对必要时才执行实际的拷贝。Swift 管理所有的值拷贝以确保性能最优化，所以你没必要去回避赋值来保证性能最优化。

上一篇: 枚举 (/project/swift/chapter2/08_Enumerations.html)

下一篇: 属性 (/project/swift/chapter2/10_Properties.html)